Types

- Types are a way to classify terms (programs).
- Meaningful terms (e.g., those that do not get stuck) should have a type.
- A typing relation relates terms to types.
- Two ways to define semantics:
  - Curry-style: Define terms and their semantics, then define types to reject those terms whose semantics are problematic.
  - Church-style: Define terms and a typing relation, then define semantics only for well-typed terms.
Typed arithmetic expressions

\[
t ::= \text{true} \quad \text{Terms} \\
| \text{false} \\
| \text{if}(t, t, t) \\
| 0 \\
| \text{succ } t \\
| \text{pred } t \\
| \text{iszero } t
\]

\[
T ::= \text{Bool} \quad \text{Types} \\
| \text{Nat}
\]

Typing relation for arithmetic expressions

The smallest binary relation “::” between types and terms satisfying all instances of the following inference rules:

- **T-TRUE**
  \[
  \text{true} : \text{Bool} \quad \text{T-TRUE}
  \]

- **T-FALSE**
  \[
  \text{false} : \text{Bool} \quad \text{T-FALSE}
  \]

- **T-IF**
  \[
  t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T \\
  \text{if}(t_1, t_2, t_3) : T
  \]

- **T-ZERO**
  \[
  0 : \text{Nat} \quad \text{T-ZERO}
  \]

- **T-SUCCE**
  \[
  t_1 : \text{Nat} \\
  \text{succ } t_1 : \text{Nat}
  \]

- **T-PRED**
  \[
  t_1 : \text{Nat} \\
  \text{pred } t_1 : \text{Nat}
  \]

- **T-ISZERO**
  \[
  t_1 : \text{Nat} \\
  \text{iszero } t_1 : \text{Bool}
  \]
Properties of the typing relation

A term $t$ is said to be well-typed if there is a type $T$ such that $t : T$.

- **Uniqueness of types:** Each term $t$ has at most one type $T$ such that $t : T$.
- **Progress:** For every well-typed term $t$, either $t$ is a value or there is a $t'$ such that $t \rightarrow t'$.
- **Preservation:** If $t : T$ and $t \rightarrow t'$ then $t' : T$.
- Safety = Progress + Preservation

Simply-Typed $\lambda$-Calculus

Enriched $\lambda$-Calculus

- Recall booleans, numbers and operations on them can be encoded in the pure $\lambda$-calculus
- Nevertheless, it is convenient to include primitive data types in the calculus as well
- $\lambda B$ is an enriched calculus with boolean data types $\text{true}$ and $\text{false}$, and operation $\text{if}$.
  \[ \lambda x. \lambda y. \text{if}(x, y, x) \] is a term in $\lambda B$.
- $\lambda NB$ is a similarly enriched calculus with numbers and booleans
  \[ \lambda x. \lambda y. \text{if}(\text{iszero}(x), \text{succ}(y), x) \] is a term in $\lambda NB$
Simply-Typed $\lambda$-Calculus

Syntax:

$$t ::= \begin{array}{ll}
\text{Terms} & \\
\text{Variable} & x \\
\text{Abstraction} & \lambda x : T. \\ t \\
\text{Application} & t \ t \\
\end{array}$$

$$T ::= \begin{array}{ll}
\text{Types} & \\
\text{Base types} & A \\
\text{Type of functions} & T \to T \\
\end{array}$$

$$\Gamma ::= \begin{array}{ll}
\text{Contexts} & \\
\text{Empty Context} & \emptyset \\
\text{Variable Binding} & \Gamma, x : T \\
\end{array}$$

Evaluation (Call-By-Value)

Small-Step Evaluation Relation for simply-typed $\lambda$-calculus:

$$
\frac{t_1 \to t'_1}{t_1 \ t_2 \to t'_1 \ t_2} \quad \text{E-APP1}
$$

$$
\frac{t_2 \to t'_2}{v_1 \ t_2 \to v_1 \ t'_2} \quad \text{E-ABS2}
$$

$$
(\lambda x : T. \ t_1) \ v_2 \to [x \mapsto v_2]t_1 \quad \text{E-APPABS}
$$
Simply-Typed \( \lambda \)-Calculus

Typing Relation

\[
\begin{align*}
\frac{x : T \in \Gamma}{\Gamma \vdash x : T} & \quad \text{T-VAR} \\
\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1. \ t_2 : T_1 \to T_2} & \quad \text{T-ABS} \\
\frac{\Gamma \vdash s : T_1 \to T_2 \quad \Gamma \vdash t : T_1}{\Gamma \vdash (s \ t) : T_2} & \quad \text{T-APP}
\end{align*}
\]

Properties of the typing relation

A term \( t \) is said to be \textit{well-typed} in context \( \Gamma \) if there is a type \( T \) such that \( t : T \).

- **Uniqueness of types:** In a context \( \Gamma \), each term \( t \) has at most one type \( T \) such that \( t : T \).
- **Progress:** For every closed, well-typed term \( t \), either \( t \) is a value or there is a \( t' \) such that \( t \to t' \).
- **Preservation under substitution:** If \( \Gamma, x : S \vdash t : T \) and \( \Gamma \vdash s : S \), then \( \Gamma \vdash [x \mapsto s]t : T \)
- **Preservation:** If \( \Gamma \vdash t : T \) and \( t \to t' \) then \( \Gamma \vdash t' : T \).
- **Safety = Progress + Preservation**
Erasure and Typability

`erase` is a function that maps simply-typed \( \lambda \)-terms to untyped \( \lambda \)-terms.

\[
\begin{align*}
\text{erase}(x) &= x \\
\text{erase}(\lambda x : T. \ t) &= \lambda x. \text{erase}(t) \\
\text{erase}(t_1 \ t_2) &= \text{erase}(t_1) \ \text{erase}(t_2)
\end{align*}
\]

- If \( t \rightarrow t' \) under typed evaluation relation, then \( \text{erase}(t) \rightarrow \text{erase}(t') \)
- If \( \text{erase}(t) \rightarrow m' \), then there is a simply-typed term \( t' \) such that \( t \rightarrow t' \) (under typed evaluation relation) and \( \text{erase}(t') = m' \)
- An untyped term \( m \) is **typable** if there is some simply-typed term \( t \) and type \( T \) and context \( \Gamma \) such that \( \text{erase}(t) = m \) and \( \Gamma \vdash t : T. \)
- **Not every untyped lambda term is typable!**
  Example: \((x \ x)\)